FSC Designer: A Visual FSM Design Tool for Robot Control

Okan Aşık, H. Levent Akın

Boğaziçi University, İstanbul, Turkey

Abstract. This paper describes specification and design of a visual tool which enables easy finite state machine design and automatic code generation. This tool is developed to achieve quick behavior development based on finite state machines. It is specifically developed for Robocup standard platform league teams to provide easy design-code-test cycle. However, it can be used in other platforms because the visual designer module of the whole system generates a platform independent modified version of SCXML. It also provides push-button code generation and compilation in addition to development of finite state machines.

Keywords: fsm, robot control, high level planning

1 Introduction

A robot may seem like a bunch of motors and some processing units. Actually, this is true, if we just consider the hardware of a robot. Although the importance of different modules changes drastically from domain to domain, the importance of good robot software is a truth. In Robocup domain, there are variety of different robot tasks. These tasks emphasis different parts of robotics research space. For example, Robocup standard platform league struggles with the problems of autonomous multi-agent robots, and Robocup small size league specializes on high level planning. Although every Robocup league specializes in different research area, every robot software needs a planning module in different granularities. Planner module is critical in that even for very simple tests you need to develop a plan using the current software architecture of the robot. Since developing software for robot is very error prone and test-driven, we need to find a way to minimize time spent on developing robot behavior. Developing complex behaviors which rise on top of basic behaviors, the complexity increases exponentially, since it is very hard to implement a finite state machine by writing many switch and cases. In some works[9], [11], [10], finite state machine representations are successfully used for high level robot control, but they lack a graphical tool to free the designer from mangled state machine codes. We aim to free the human brain from complex switch-case based FSM coding, and achieve full efficiency of behavior designer. To solve this problem, we represent a new visual tool, FSC Designer.

Briefly, FSC Designer facilitates the development of finite state machines. It outputs the visual design as an XML file whose format is similar to State

Chart XML (SCXML)[5]. We represent the behavior as a subset of conventional FSM. Only states and transitions are used in FSC Designer and action concept is added. Robot can be only in one state, and one state can have none or many actions. If there is an action for the current state, this action is executed. Transitions are used to change the active state and condition code are written by the behavior designer.

This tool is developed on top of Cerberus[1] code base. Cerberus is one of the well-known Robocup standard platform league (SPL) teams. Since it is a SPL team, well developed simple behaviors have critical importance for success. Development of such simple behaviors requires too much testing and tuning. Therefore, FSC Designer facilitates easy planning by automating behavior development.

Firstly, we will discuss the related high level planning architectures. Then, we will present the our visual tool. We also compare and contrast our tool with the current approaches. Finally, we present the conclusion and future work.

2 Related Work

When we investigate the Robocup domain, we can categorize planning architectures in to three category: (1)Behavior-based[7], (2)Finite state machine-based[11][9][6], and (2)Logic-based[3]. Although we defined a well-specified categorization, FSM-based architectures can be viewed as a variation of behavior-based systems because they formalize behavior specification by enabling different combinations of basic behaviors. Therefore, we can view FSMs as the implicit arbitrator mechanism in behavior-based architectures.

In [7], they propose a modular hierarchical behavior-based planning architecture. There are three parts of the system, namely, sensors, behaviors, and controls. Sensors represent the state of the environment with different granularity. Behaviors define expected actions of the robot from abstract to basic motor controls. Controls are arbitrator mechanism which maps the high-level behaviors to low-level behaviors.

Although, developers of robots try to define the behavior of the robots, they are not expert of the domain they develop the robot for. Therefore, in [12], a new behavioral planner tool is presented which aims to ease behavior specification by domain experts not by developers. This tool enables specification of new behaviors based on basic behaviors and adjusting user friendly parameters of the specific behavior. This tool automatically generates codes for the robot and enables knowledge transfer from domain expert to robot.

The main motivation for using FSM in robot planning is a need for a easy to design graphical formalism. In [4], [10], [9], they propose Hierarchical Finite State Machines to achieve such formalism. [4], and [10] proposes a visual tool to ease design-test cycle of robot planning. These tools also automatically generate code to run on robots. In [9], they propose a more general robot planning framework in which they use XABSL programming language to define agent behaviors. This

tool also has capability to generate code which is ready to run on robots provided that XABSL engine exists for the particular platform.

Logic-based control is another high level planning approach. In [3], a new language READYLOG which is based on GOLOG [8] is designed. The main feature of READYLOG is making use of decision-theoretic planning which basically approximate expected utility of different plans.

Although true vision of AI is a system which can learn and adapt its behaviors according to current state of the environment, those systems generally result in poor performance for particular cases such as robot soccer. This is mainly due to need for a specific behavior for particular situations which are hard to integrate in to a learning problem. That is why, most of the planning architectures are hand-coded as investigated above.

3 FSC Architecture

The proposed planning architecture is part of a robot software architecture. It is developed on Cerberus robot framework[2]. This framework supports many different planning architectures. Planning architectures access abstract robot and uses abstract robot to use other component of the robot such as the model of environment or motion module. If there is no such central module, developing a planner architecture on top of a robot software architecture could be very hard. So it is important to note the role of such central control in extension of the current software arhictecture. All transition conditions use abstract robot to access whole state of the robot and environment. There are two parts of FSC architecture, namely, FSC Planner Engine and Visual FSC Designer.

3.1 FSC Planner Design

Our finite state controller(FSC) is modified version of well-known finite state machines. There are only three concepts in a FSC. These are states, transitions, and actions. States and transitions have the very same semantic as the conventional finite state machines. However, we removed the idea of event-based transitions since we have fine control of every planner step. Since transitions will not occur on some events, we have to develop a new method to carry out state transitions. If we assume that robot software runs its modules in some order, at some point it will also run our FSC. When the robot software runs the FSC planner, we check all transitions of the active state for a transition. If one of the transition conditions returns true for a transition, the active state changes in the next planner step according to target state of that transition. Since planner checks transitions one by one, there is an implicit transition priority which is based on the order of transitions. The latest a transition added lowest its transition.

Another important concept is action. Action should not be confused with finite state machine events since they are completely different things. Actions are simple behaviors which can run for a single planner step. Every state has none or many actions, and at every planner step first the action of the current

state is executed, then transition conditions are checked. Since states can have more than many actions, we can combine different simple behaviors for complex behaviors such as combining head and walking for ball searching behavior.

3.2 FSC Planner Engine

FSC Engine is a particular implementation of the proposed approach for specific robot software architecture. We already implemented a FSC Engine on top of Cerberus code base. FSC Engine provides interface to realize visually designed finite state controller. There are four critical concept in this module. They are FSC Planner, FSC, state, transition and action. In code level, these concepts corresponds to particular classes. FSC Planner Engine takes the XML representation of FSC and by extending engines base classes implements the intended FSC. By developing different FSC Planner engines for different robot platforms, we can design behaviors once and can use it in many different robots.

3.3 Visual FSC Designer

Visual FSC Designer is the front-end of the proposed system. It provides a easy to use graphical design tool. It has three functionalities:(1)graphical representation and manipulation of finite state machines, (2)code generation, and (3)robot software compilation. In the domain of finite state machine, there are many graphical tools to facilitate software engineering process, but all these tools directly addresses the problems of software engineering domain. Visual FSC Designer can be thought as specialization and extension of these tools for the robot control case.

As it can be seen in 1, there are 5 parts of the tool: Menu bar, toolbox, scene, property browser, and object browser. Menu bar provides easy access to functionality of the tool such as opening a FSC, creating a new FSC, or generating code. Toolbox carries scene elements, such as State and Transition. Scene is the design area of the tool. FSC Designer provides a drag-and-drop fashion finite state machine design. We can move objects on the scene or change their ids. Property browser provides an easy to use property management for state and transition objects on the scene. When an object is selected via the scene or object browser, the properties of the object are listed in the property browser. Although this property browser seems a restricted environment to change some properties transition condition code or actions of a state, it popes up customized property environment for particular properties as it seen in 2. Object browser is a facility to ease the object access in case of hard object manipulations on the scene. Since all objects are physically on the scene, for complex finite state machines, some objects may interfere and block other objects because of limited scene view. Also, some behavior designer may prefer directly access the objects without dealing with the scene. In all these cases, objects can be accessed by using object browser.

Code generation is another facility of FSC Designer. Code generation system takes the XML representation of the designed finite state machine and generates

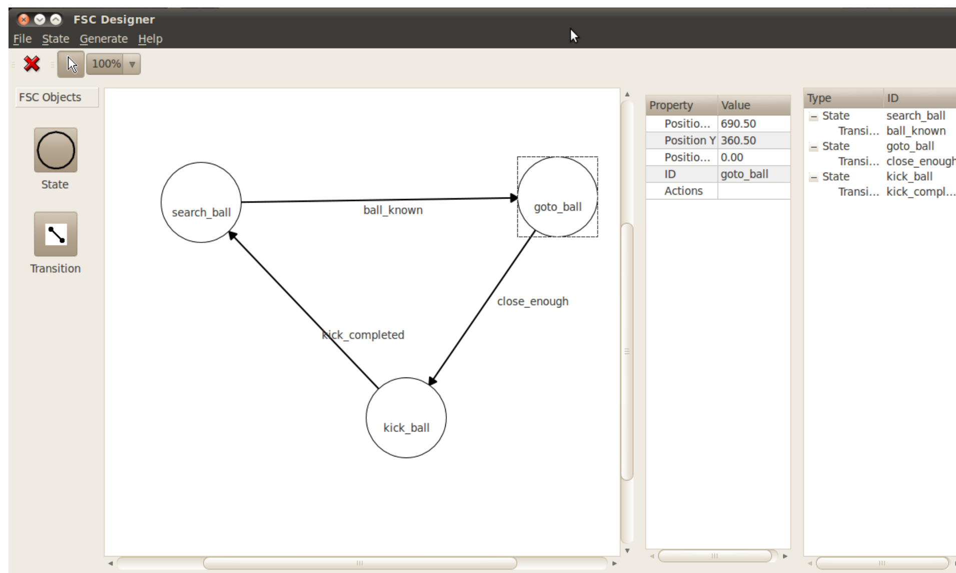

Fig. 1: A screenshot of the FSC Designer

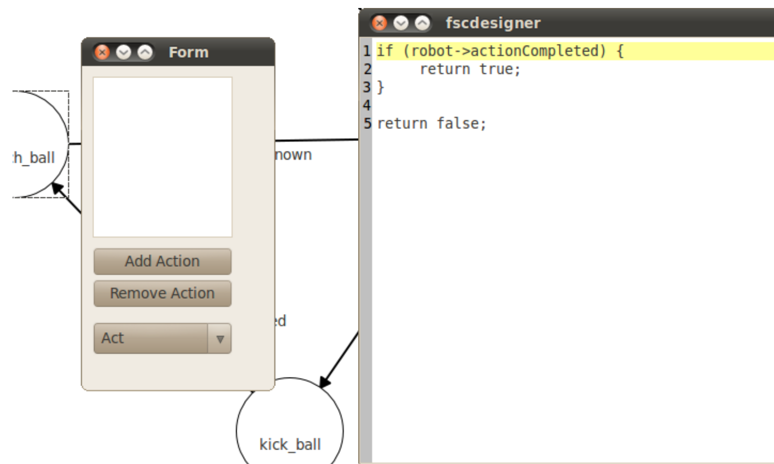

Fig. 2: Customized Property Editors: Left: Action Selector, Right: Code Editor

the code for the target robot software. Code generation part is developed so that it can produce finite state controller provided an XML representation and particular FSC Planner Engine. In the implementation level, we used a template-based code generation technique. We developed three templates: (1)finite state controller, (2)state, and (3)transition. We developed our templates so that every state and transition extends the base classes of the target robot platform. Because of this relation with the base class of the target robot platform, code generation templates are robot platform specific. Actions are also robot platform specific since they are the basic building block of the whole behavior of the robot.

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0" initial="SearchBallState">
    <state id="SearchBallState" x="369" y="503">
        <transition id="BallKnownT" event="" cond=" if (robot.ballknown) return true;"/>
        <action id="SearchObject"/>
    </state>
    <state id="KickState" x="804" y="455">
        <transition id="BallUnknownT" event="" cond="if (!robot.ballknown) return true;"
    target="SearchBallState"/>
        <action id="Kick"/>
    </state>
    <state id="GoToBallState" x="506" y="709">
        <transition id="CloseEnoughT" event="" cond="if (robot.ball.distance < threshold)
    return true;" target="KickState"/>
        <action id="GoToPose"/>
    </state>
</scxml>
```

Fig. 3: An example XML representation of a finite state machine

Another important facility of FSC Designer is push-button code compilation. If we conceptualize the behavior development phases, as design, code generation, compilation, and test. It is critical to ease the compilation of new designed behaviors. To this step, we already generated related codes, we just need to compile the code and run it on the robot. Compilation is again the robot platform dependent part of the system, but any robot software platform using Makefiles can be integrated to the FSC Designer compilation module. Actually, any thing which can be compiled using unix command line utilities can be integrated with little modification.

4 Discussion

The system we proposed in this paper may seem a simple engineering, but it is not true. The experiences in the Robocup has proven that many teams work

hard on the basic behaviors for success, but they run out of time to develop a high level behaviors out of these basic behaviors, especially in leagues which require well-engineered basic behaviors such as standard platform league. Our system solves this problem by providing a simple graphical tool for high level behavior design.

There are some similar works mentioned in the related work section. However, some of them([12], [4]) is so restrictive to develop something really complex, some of them([9]) are so much complex to ease the process of behavior development, even some require learning a new scripting language. Our work tries to fill the gap between these two approaches.

5 Conclusion and Future Work

The proposed system can be used for different robot platforms since they all share some common software patterns and portable design of FSC Designer. Visual FSC designer does not need any change, but other platforms should develop their own specific implementation of finite state controllers. Although developing a specific implementation of FSC seems time consuming, the payoff is great.

The current implementation of finite state controller does not utilize the hierarchical finite state machines, but the system is supposed to support hierarchical finite state controllers for next versions.

There is also limited support to write transition conditions correctly. There is a need for a good syntax checker to be able to write error-free transition conditions. First version XML representation is some modified version SCXML, for better compatibility, it is better to represent our finite state controllers with SCXML.

References

1. http://robot.cmpe.boun.edu.tr/ cerberus/wiki/
2. Akın, H.L., Gökçe, B., Özkucur, E., Kavaklıoğlu, C., Sevim, M.M., Ayar, T., Aşık, O.: Cerberus'11 team description paper. Tech. rep. (2011), http://robot.cmpe.boun.edu.tr/ cerberus/wiki/uploads/Downloads/Cerberus2011-TDP.pdf
3. Ferrein, A., Lakemeyer, G.: Logic-based robot control in highly dynamic domains. Robot. Auton. Syst. 56, 980–991 (November 2008), http://portal.acm.org/citation.cfm?id=1453261.1453486
4. Hugel, V., Amouroux, G., Costis, T., Bonnin, P., Blazevic, P.: Specifications and design of graphical interface for hierarchical finite state machines. RoboCup 2005: Robot Soccer World Cup IX pp. 648–655 (2006), http://www.springerlink.com/index/e6hkp3t810g88239.pdf
5. Jim Barnett, G., et al et al: State chart xml (scxml): State machine notation for control abstraction (2010), http://www.w3.org/TR/scxml/
6. Lausen, H., Nielsen, J., Nielsen, M., Lima, P.: Model and behavior-based robotic goalkeeper. RoboCup 2003: Robot Soccer World Cup VII pp. 169–180 (2004), http://www.springerlink.com/index/RN06QL9N3QYMDD3D.pdf

7. Lenser, S., Bruce, J., Veloso, M.: A modular hierarchical behavior-based architecture. In: RoboCup-2001: The Fifth RoboCup Competitions and Conferences. pp. 423–428. Springer Verlag (2001)
8. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: Golog: A logic programming language for dynamic domains. Journal of Logic Programming 31 (1997)
9. Loetzsch, M., Risler, M.: XABSL-a pragmatic approach to behavior engineering. of IEEE/RSJ International Conference of pp. 1–6 (2006)
10. Murray, J.: Specifying agent behaviors with UML statecharts and StatEdit. RoboCup 2003: Robot Soccer World Cup VII (2004), http://www.springerlink.com/index/903XBPAQWU5HKRVB.pdf
11. Obst, O., Obst, O.: Specifying rational agents with statecharts and utility functions (2001)
12. Scerri, P., Coradeschi, S., Törne, A.: A user oriented system for developing behavior based agents. In: RoboCup-98: Robot Soccer World Cup II. pp. 173–186. Springer-Verlag, London, UK (1999), http://portal.acm.org/citation.cfm?id=646583.698084